並行処理による数式処理の試み(数式処理と数学研究への応用)

AUTHOR(S):

野呂, 正行; 竹島, 卓

CITATION:

野呂, 正行 ...[et al]. 並行処理による数式処理の試み(数式処理と数学研究への応用). 数理解析研究所講究録 1989, 685: 146-151

ISSUE DATE:

1989-03

URL:

http://hdl.handle.net/2433/101208

数理解析研究所講究録
第 685 巻 1989 年 146-151

並行処理による数式処理の試み

富士通国際研
野呂正行　竹島卓
(Masayuki Noro,
Taku Takeshima)

1. 数式処理における並行処理

数式処理において重要な手段の一つに Hensel 構成がある。多くの場合、Hensel 構成は、
法の選択 ⇒ タネ、bound の計算 ⇒ 実際の構成 ⇒ テスト
というふうに行われ、最後のテストに失敗した場合、最初に戻って法の選択からやり直す。この場合、法は unlucky であったといわれる。また、因数分解の場合、法によっては、タネの個数が特に多くなり、構成に無闇に時間がかかる場合もあり、この場合も法が unlucky であったといってよい。このような unlucky な法を避けるために、従来、あらかじめ幾つか (例えば 5 個) の法を選び、それらについてタネを計算し、それらの中で、最も都合のよいものを選んでその後の計算に使う、ということが行われてきている。しかし、このような方法では、結局実際には使われなかったタネを計算するために使われた計算時間はほぼ完全に無駄になる。この場合、タネの計算を幾つか並行して行うことができれば、(実際に CPU が複数個使用できる場合には) 高速化できる。また、あらかじめ選んだ法の中に lucky なものが必ずあるとは言えないため、結局やりなおしということもありうる。これは、特に GCD 計算の場合には致命的である。この場合、タネの計算を並行して行うことができれば、最悪の場合は避けられる。(ただし、この場合、Hensel 構成に使用中の法が unlucky であることが判明した場合、直ちに通知できることが必要である。) 後述するが、Hensel 構成に限っても、これ以外にも並列化により高速化できる部分がある。
以上述べたように、数式処理においては、vector に対する並行処理などのように明らかな応用の他にも幾つかの応用が考えられる。しかも、これらは真に実用を目指すもので、並列のための並列といった類のものではない。

2. 並行処理を可能とする環境

並行処理を行うためには、複数の process が同時に走り、かつ互いに通信しあえることが必要である。これらについて分けて述べる。

複数 process の同時実行

これは、processor が複数台あれば当然である。1 台のみであっても、TSS などにより multi-process が実現されていればよろしい。(ただし、1 つの process が動く時、他の process が長時間 stop してしまうようなものは論外である。)

process 間通信

大きく分けて、stream による通信と、shared memory による通信とが考えられる。実際の

効率は shared memory によるものの方が高いであろうが、通信 data が不定長ということと、異なる machine 上での shared memory は multi-processor をサポートした OS 上でしか利用可能でないため、UNIX 上では stream による通信を使うことになる。

UNIX の場合、効率、速度の点でやや不満は残るものの、一応上の二つの条件は満たしている。また、TCP/IP による process 間通信は、同一 machine あるいは異なる machine 上の process 間の通信を区別なく可能にするから、workstation が複数台使用できる状況に適している。(実際には、stream は byte-stream であるため、machine によっては byte-order が問題となる場合もある。しかし、XDR protocol をサポートしている machine どうしではこの問題は生じない。)
上述の条件が満たされれば、最小限並行処理は実現できる。しかし、結果が出た時点で直ぐに他の process に通知する機能がなければ効果は半減する。UNIX 4.2BSD 以降の版では、これを実現するものとして out-of-band data (OOB と略す) を用意している。この指定を行って他の process に message を送ると、受け手の process は message を SIGURG (urgent signal) と共に受け取る。よって、正しく signal handler を setting しておけば、上の機能は実現できる。
以上のことから、現在個々の機能(因数分解、GCD その他)を作成中の UNIX 4.2BSD (SUN OS 3.x) 上で並行処理をある程度試みることが可能なことが分かる。

3. 実現する上での問題点

UNIX 上で process 間通信による並行処理を行うとき、幾つかの問題点が生じる。それらについて述べる。

通信の overhead

stream による通信は、本質的に data の copy だから、通常の函数呼び出しの場合の pointer 参照に比べて既に効率が悪い。更に、read、write などは system call であるため、これらを小刻みに繰り返すと全体の performance の低下を招く。これを避けるためにはなるべく buffering をして一度に読み書きする必要がある。

非同期性による問題

以下、主 process (main と呼ぶ) は一つだけ存在し、並行して走る process (sub と呼ぶ) たちは、全て、main から command 、data を受け取り、結果を main に返す、という動作を繰り返すものとする。main は、幾つかの process が同時に結果を返してくる場合には、select() system call により結果の到着している socket を調べることができる。ここまでの段階では非同期性による問題は生じない。問題は何らかの理由で緊急に message のやりとりを行う場合である。例えば main が、ある sub から受け取った data を他の sub に転送する場合、受け手が read 中の場合には、完全に独立な緊急 message 用の channel を持たない限り、read 中の data と緊急 message の区別がつかない。また、受け手が write 中で

あった場合には、同様の理由で返事を返すことはできない。(上述の out-of-band data は、implement にやや問題があるらしく、同期信号としてのみ使用している。) また、同期信号である OOB も結局は stream による送受信となるため、受け手側に遅れが生ずる場合もあり得る。更に、複数個の signal が同時にやってきた場合に、取りこぼしの問題も生ずる。
この問題を回避する方法は、状況により異なるが、最小限次の制約を設ける。

1) main、sub 共に、受け付ける緊急 signal は一度に一個。送り手は、応答があるまで次の緊急 signal を送れない。

2) 緊急 signal により global に jump できるのは、main、sub いずれか一方とする。

3) 緊急 signal による global jump 後の normal read により受け取る data は受け手にとって素性が明らかでなければならない。

debug

multi-process の debug は一般に困難である。特に main から起動される sub の場合はそれ自身制御端末を持たないためより困難を伴う。SUN の場合、debugger により走行中の process に attach できるため、この困難は回避できる。とは言え、一般に program は非決定的に動作するため、sub それぞれの debug 用情報を逐一知る必要が出てくる。このために、X-Window との interface を作成し、debug 用情報を monitor できるようにしている。

4. 実行例

4-1. 一変数多項式の因数分解

もともとのアルゴリズムは次の様になっている。入力多項式を f (無平方) とする。

1) 幾つかの法 p を選び、f mod p の Frobenius 行列の rank を計算し、最も rank の大きいものを選ぶ。
2) 1) で選んだ p で、有限体上の因数分解を行う。
3) Mignotte bound を求め、parallel lifting を行う。
4) trial division を行って、真の因子を求める。

幾つかの例について、これらの step 毎の timing data を図 1 に示す。これらの内、明らかに 1) が並行化可能である。従来この step は、p を 5 個選んで sequential に実行していた。図 1 から分かるように、1) は、bottle neck ではないが、無視できない程度の時間を消費する。1) を並行化したものと、従来のものとの比較をを図 2 に示す。時間は、main における cputime + max(sub の cputime) で、I/O の待ち時間は含まない。これでみる限り、実際に speed が上がったようであるが、CPU が 2 台しかないため (しかも 1 台はもう 1 台の半分の speed)、実時間ではほとんど差がなかった。また通信の overhead のため小さな問題では、かえって遅くなることもある。また、この例では、どの場合も同程度の時間で

計算が終了するので動的な制御による劇的な効果は出にくい。なお、X-Window 上での実行イメージを図3に示す。
さて、もう一つの可能性は、3) における、真の因子の早期発見である。parallel lifting では、Wang の方法では、早期発見は lifting 中の各候補についてのみ行うが、これを、lifting と並行して sub に 4) を行わせることによってより完全なものにする、という方法も考えられる。これについては現在試作中である。

4-2. EZ-GCD

EZ-GCD は、evaluation、一変数での gcd、hensel 構成、test からなるが、evaluation が unlucky であった場合、それが判明するのは test を行った後であり、それまでの Hensel 構成にかかる cost が極めて大きい。この場合、evaluation+一変数での gcd を、Hensel 構成と並行して行うことにより、unlucky evaluation を早期発見できる。
伝統的な無平方分解においては、GCD(f, f')の計算が最も時間がかかる。この部分を並行化した実行例を図4に示す。この例で分かるように、evaluation point の選び方によっては、unlucky evaluation がかなり続く可能性があることが分かる。

終わりに

これまで、主に具体例を通して述べてきたが、より汎用的かつ安全なものにするためには、更に実例を増やして考察することが必要である。最終的には、他の部分と同様 library として black box 化するのが望ましい。また、process 間通信の応用という点では、鈴木、佐々木他の ANS とも関連するが、parser(interpreter) と本体を process 間通信で結ぶ、というアイディアもある。すなわち、本体は、Xserver などと同様、primitive な仕事のみ行い、その結果を整理活用するのが parser である、と分離するのである。この方式の利点は、本体を access する library さえ完備しておけば、parser としては、lisp でも prolog でもなんでも使用できることである。欠点は、通信量が膨大なものになる恐れがあることだが、頻繁に使用する部分は compile して本体に組み込めるようにすればこれは解決する。

以上、主に UNIX の process 間通信機能を用いた並行処理について述べてきた。しかし、UNIX の process 間通信は、使い易さ、機能、効率の点で今一歩である。この面での UNIX の改良が望まれる。また、TCP/IP さえきちんとサポートしていれば、大型機を用いた並行処理、あるいは work station をフロントエンドとする分散処理も可能である。もしこれが実現すれば、数式処理もより一層使いやすいものとなるであろう。

	f1	f2	f3	f4
step1	3.37	0.78	0.30	6.05
step2	1.35	0.43	0.18	9.68
step3	3.43	1.60	0.23	32.9
step4	0.58	0.42	0.05	135
total	9.00	3.35	0.87	184

図1 (単位: 秒)

	f1	f2	f3	f4
parallel-1	1.15-1.40	0.27-0.35	0.22-0.32	2.40-2.50
parallel-2	1.20-1.30	0.32-0.33	0.18-0.23	2.43-2.75
parallel-3	1.77-2.13	0.42-0.55	0.20-0.32	3.67-3.70
parallel-4	2.20-2.38	0.77-0.98	0.27-0.48	3.00-3.28
sequential	3.37	0.78	0.30	6.05

図2 (単位: 秒)

$$
\begin{aligned}
f1 =&(4096x^{10}+8192x^9-3008x^8-30848x^7+21056x^6+146496x^5\\
&-221360x^4+1232x^3+144464x^2-78488x+11993)\\
&\times(4096x^{10}+8192x^9+1600x^8-20608x^7+20032x^6+87360x^5\\
&-105904x^4+18544x^3+11888x^2-3416x+41)\\
&\times(8192x^{10}+12288x^9+66560x^8-22528x^7-138240x^6+572928x^5\\
&-90496x^4-356032x^3+113032x^2+23420x-8179)\\
&\times(8192x^{10}+20480x^9+58368x^8-161792x^7+198656x^6+199680x^5\\
&-414848x^4-4160x^3+171816x^2-48556x+469)\quad\text{(SIGSAM problem 7)}
\end{aligned}
$$

$$
\begin{aligned}
f2 =&(123456789x^2+9876543x+5)(100000000001x^2-89x+234565431)\\
&\times(256x^3+5x^2+798000)(3125x^3-x-125)(256x^6+x^3-2000x^2-3125x-8625)
\end{aligned}
$$

$$
f3 =2(234x^5+98x^4-1234x+8)(192837x^5-x^3+1287x^2-1)(999x^3+765x^2-x-1)
$$

$f4 = resultant(f(x-2y), f(y), y)$ $(f(x)$ は、$\sqrt{2}+\sqrt{3}+\sqrt{5}$ の最小多項式 (8 次))

使用した計算機は、SUN3-260(4MIPS) と SUN3-110(2MIPS) で、図2における parallel-1 から 4 は、次の通りの構成。

parallel-1 : 260 に process が 5 本
parallel-2 : 110 に 1 本、260 に 4 本
parallel-3 : 260 に 4 本、110 に 1 本
parallel-4 : 260 に 3 本、110 に 2 本

2 と 3 では、main から data を渡される順序が異なる。

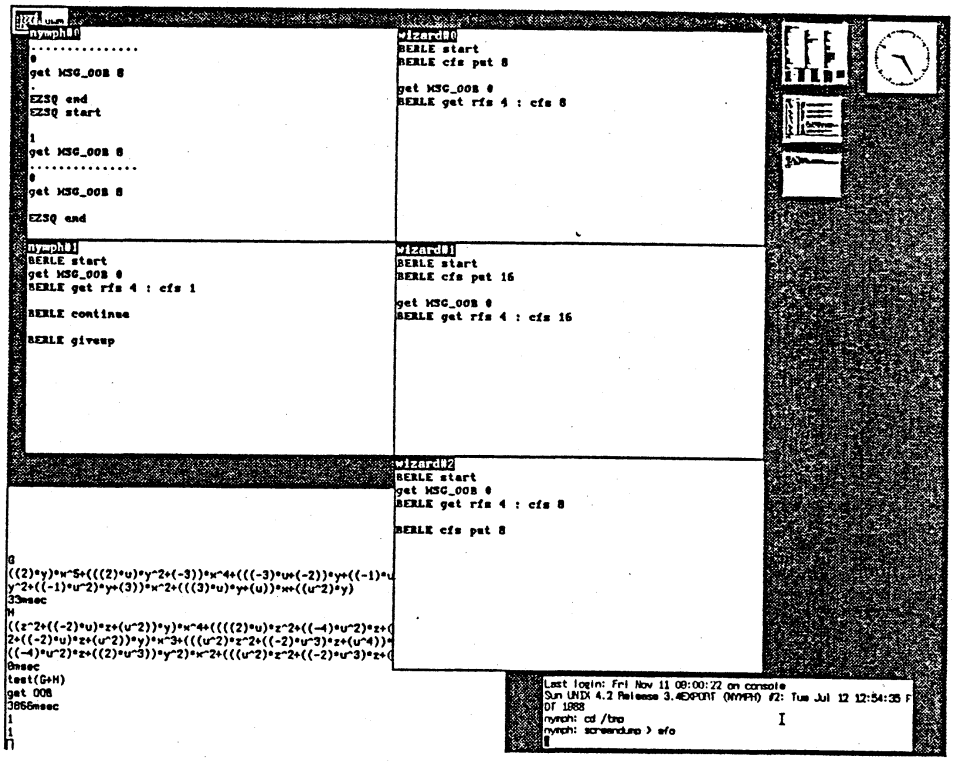

図3

図4

$G = (x^2-1)(uy+x)(2x^2y-3x-u)$

$H = xy(x+1)(uy+x)^2(u-z)^2$

	G	H
parallel	1(0.28)* -3 回 - 0(0)	2(0.02)* 1(0.33)
old	1(3.72)0(0)	2(2.45)1(0.28)

	G^2	H^2
parallel	6(0.32)* -3 回 - 5(5.32)	6(0.32)*5(2.58)
old	6(82.3)5(4.57)	6(31.5)5(2.33)

	$G+H$	G^2+H
parallel	1(3.82)* -15 回 - 0(0)	2(1.38)*1(8.22)
old	1(130)0(0)	2(1095)1(7.82)

$n(m)$: $deg(gcd(f(x,b), f'(x,b))) = n$ (m秒)

*付きは、sub により中断されるまでの時間 (秒)

-n回 - : sub で失敗がn回